All about functions

Prof. Zhang

April 7, 2014

1 Pre-defined function

Programmers do not build everything themselves. Instead, they try very hard not to reinvent the wheel. The first language mechanism to facillate code reuse is **functions**. A **function** packages a piece of codes with a well-defined functionalities, gives it a name, and specifies the input and output of the function.

For example, the function **pow** is implemented in the math library (each library can be thought of as a collection of related functions). It takes the base and exponent as input, and return the power (base raised to the power of exponent) as output.

```
double pow (double base, double exp);
```

Online manual You can use command **man** to view online manual for pre-defined functions (implemented in libraries) (and also commands). For example,

```
$man pow  //to learn more about pow function
$man ls  //to learn more about the ls command
```

Exercises Learn about function **srand()** using **man** command.

1.1 Function call syntax

The following is the syntax for making a function call in your code:

```
function_name (argument_list)

where argument_list is a comma-separated list of arguments, i.e.,

argument1, argument2, ..., argumentn
```

In your code, you can **call** or (invoke) a function as follows:

```
pow(2,31)
```

1.2 Using function call in expression

If a function returns a value, you can use it in an expression. Since the function **pow** returns a double type of value, you can use the function anywhere a double type of value can appear:

```
num/power(2,i)+pow(4,2)
cout << pow (3,9);
```

2 Build your own function

You can **define** your own function, so that you can use it again and again (to do something different by varying the arguments):

2.1 Syntax: Function Definition

```
type_returned function_name (parameter_list)
{
   statement1;
   ...
   statementn;
}

where type_returned can be void, int, double, char, bool, string, ...

parameter_list is a comma-saparated list of parameters:

type1 para1, type2 para2, ..., typen paran
```

Note that we have names for the different parts of the function definition:

- **function header**: the first line of function definition, where the return type, function name, and parameter list are specified.
- **function body**: the statements enclosed by { and }. Any statement can be put in a function. Variables declared in a function are local variables.

For example, the following segment define a function that returns an **int** type value to the caller.

```
int sum (int num1, int num2)
{
   int total=num1+num2;
   return total;
}
```

2.1.1 The return statement

If a function's return type is not **void**, then it must use **return statement** to return a value to the caller (which can be the main, or other functions).

Syntax of **return** statement:

```
return expression;  //the value of expression must of be "return type"
       // ends the function call, and returns the value of expression
       // to the caller.
or
return;   // used for void function
          // ends the function call, i.e., return control to the caller...
```

2.1.2 void function or procedure

Another name for a function with **void** return type is **procedure**.

The following function works only in Mac machines, which have a **say** command (http://www.gabrielserafini.com/blog/2008/08/19/mac-os-x-voices-for-using-with-the-say-command/). The function "says" the message, no value is returned, so we use keyword **void** as its return type.

```
/* this function say the given message
void SaySomething (string message)
{
   string command;
   command = "say "+ message;
   system (command.c_str());
}
```

2.2 Program Structures with functions

In most programming languages, a name (i.e., identifier) must first be **declared** before being used. For example, a variable must be first be declared before you can access it. Likewise, a function must be first **declared** before being called.

There are two possible ways to organize your programs.

1. Function Definitions Before main We can define all functions before our **main** functions. For example,

```
#include <iostream>
using namespace std;

int sum (int a, int b)
{
```

```
   return a+b;
}

void printLine (char ch, int len)
{
   for (int i=0;i<len;i++)
     cout <<ch;
  cout <<"\n";
}

int main()
{
 ...
}
```

2. Function Definitions After main Or, we can **declare functions first**, followed by **main** function, and then **function definitions**.

The function declaration (or function prototype) only specifies the function header, i.e., function prototype, with return type, function name, and paramter list. The name of the paramters are optional. For example, the following are function prototypes:

```
void PrintTriangle (char, int);
void PrintRectangle (char ch, int length, int height);
int Max (int num1, int num2, int num3);

#include <iostream>
using namespace std;

//function declaration, i.e., function prototypes
// This provides enough information for compiler to
// check function calls...
int sum(int a, int b); // alternatively, int sum (int, int);
void printLine (char ch, int len);

//main funciton
int main()
{
 ...
}

//Function definitions, i.e., the actual things...
int sum (int a, int b)
```

```
{
   return a+b;
}

void printLine (char ch, int len)
{
   for (int i=0;i<len;i++)
     cout <<ch;
  cout <<"\n";
}
```

3 Function Call Details

3.1 Entering function

1. Passing parameters
2. Getting local variables ready

Note on Type casting syntax. The following are three possible ways to convert a value to different type:

```
static_cast<double> num
double (num)
(double)num
```

3.2 Returning from function

The control returns from function to the caller whenever 1) a return statement is executed, or 2) the function body ends (i.e., the last statement in function body has been executed.

1. If **return** is followed by an expression, the expression is evaluaed, and the value is passed to the caller
2. All local variables go out of scope (erased from memory)

A Common logic error is to call a function that returns a value, without using the return value:

```
...
sum (10,20);
pow (2, 8);
```

The above two function calls have no effects: even the functions are called and executed, the returned values are not used anyway.

3.3 Function: Putting it all together

Tracing program that have function calls. By tracing a program, we mean follow the execution of a program with a set of sample input values, in order to identify logic errors, understand codes written by others, or to verify your design and code.

The keys to program tracing:

- **Remembering where you are using Arrows**: draw an arrow to point to the current statement in each function that is currently running.

  1. In the begining, a single arrow points to the first statement in **main** function.
  2. Move the arrow within a function, following the logic structure (if/else statement, loop statement, and switch statement).
  3. When reaching **return** statement in **main**, or end of main's }, the program ends.

- **Keeping track of variables value** Each variable has a **name**, **type**, **value** and **scope**. We have learnt to use a box, labelled with the name to represent a variable. Now in order to differentiate variables of different scopes, we use **global variable tables** and **local variable tables**.

  For each **function call**, a separate memory block (called **call stack**) is used to store its local variables, which include parameters.

  At a function call,

  1. A call stack is allocated, i.e., a new set of local variables are allocated
  2. Passing parameters: actual arguments are passed into the function, meaning, actual arguments are assigned to formal parameter variables (based upon order).
     - **pass-by-value**: the **values** of arguments are assigned to parameters. The parameter (which is a local variable) is initialized with the value of the arguments.
     - **pass-by-reference**: the **address** of arguments. are assigned to parameters. **All reference to the parameter are made to the acutal arguments.** It's like substituting all occurence of the pass-by-reference parameter by its acutual argument.
  3. Remember where to return when this function finishes, i.e., **return address**.
  4. Pass control: start to execute function body from first statement
  5. Return from function: function's execution ends whenever a return statement is executed, or the end of function body is reached. If a value is returned, the value is returned to the caller. The execution resumed from the next statement in the caller function. The call stack for the function call is "popped" (i.e., deleted from memory), all local variables are forgotten.

Exercise in function design
Design the following functions, write the function header and comment for each:

1. A function that checks whether a given date is valid or not.

```
/*
 * check date's validity: return true if the given date is valid, and false otherwise
 * Pre-condition: month, day, year stores a date
 * Post-condition: true or false is returned
 */
bool ValidDate (int month, int day, int year);
```

2. A function that takes the customer's order (the number of small pizza and large pizza), i.e., main program call this function to obtain an order.

```
/*
 * Take the user's input about the number of small and large pizzas to order
 * Pre-condition: none
 * Post-condition: small_pizza stores the number of small pizza as entered,
                   large_pizza stores the number of large pizza as entered.
 */
void TakePizzaOrder (int & small_pizza, int & large_pizza);
```

3. A function that calculates the sale price, given the original price and discount rate.

4. A function that simulates rolling a pair of dice. The function should yields two random values between 1 and 6.

5. A function that reads a sequence of positive integers (end with -1) from keyboard, and "yield" to the caller the average, sum, min, and max of these numbers.

Exercise in program tracing

1. **Function with both types of parameters**

```
void AddTax (double & amt_due, double rate);

int main ( )
{
   double total_order = 120.00;
   double rate = 0.05;
   cout <<Total order:  << total_order <<endl;
   AddTax (total_order, rate);
   cout <<With tax total due: << total_order<<endl;
   cout <<Tax rate is : << rate << endl;
}

void AddTax (double & amt_due, double rate)
{
  rate = rate+1.0;
  amt_due = amt_due*rate;
}
```

2. **Recursive Function**

```
int main()
{
  cout << func (3);
}
```

```
int func (int n)
{
   if (n==1)
      return 1;
   else
   {
      int result = n * func (n-1);
        return result;
   }
}
```

4 Design Aspects

4.1 Blackbox Analogy

A function is a self-contained entity. The caller (user) of a function only needs to know what the function does (through the function comments, online manual, etc.), and does not need to know how the function works. In another word, the implementation details are hidden away from the user.

Function comments are like contracts which specify what can be expected from the function (a contractor that promises to get some work done for you): given the inputs (parameters), what outputs can be expected. The outputs of a function can be returning value, or delivered to output device (terminal, sound card, printer). More formally, the function comment shall have the following information documented:

1. A short description about what the function does
2. Precondition: What shall be true before the function call, including the parameter values, and possibly some global variables's value
3. Post condition: What shall be true after the function call, including the return value, the modification of the pass-by-reference parameters, and side-effects of the function, for example, modification of global variables

Exercises Please draw a diagram below to enforce your understanding of the concepts: use a rectangle to represent the funciton, arrows pointing to the box to represents the inputs, and arrows leaving the box to represents the returning value. Outputs sent to output devices are usually documented.

Design your function to perform a well-defined, self-contained job. This allows the maximum reuse opportunity. For example, instead of

```
// a function that prompts the user to enter a date, and check its validity,
// repeat until date entered is valid...
void EnterValidDate(...);
```

We prefer to build a function as follows, which gives us more flexibility in terms of where the funciton can be used.

```
bool IsValidDate (int year, int mon, int day);
```

To enforce the blackbox analogy, avoid design a function with so called **side-effects**. By all means, avoid modifying global variables in a function.

Exercise Write down benefits of functions (i.e., modular design). Try to be as complete as possible.

5 Variable Scope

Scope of a variable refers to the parts (context) of the program where the variable can be referenced (or is visible). The variable's scope is decided based on where the variable is declared.

- **Local variables** are variables declared in a function body. They are visible from the point where they are declared to the end of the funciton.
- **Block variables** are variables declared in a block (i.e., a block statement). They are visible from the point where they are declared to the end of the block. For example,

```
...
int sum=0;
for (int i=1;i<10;i++) //i is block variable only visible within the for loop
{
   sum+=i;
}
```

- **Global variables** are variables declared outside of any functions. They are visible from the point where they are declared to the end of the entire program.

  The most common use of global variables are for define named constants. One should try to avoid using global variables for other purposes, if possible.

```
const int COKE_PRICE=250;
const double SMALL_PIZZA_PRICE=13.99;

void PrintReceipt (int small, int large)
{
   // access SMALL_PIZZA_PRICE ...
}

int main()
{
   ...
   double total_due = ... // access SMALL_PIZZA_PRICE ...
   ...
}
```

6 About Reuse Names

Functions with same name. You can define multiple functions with the same name, as long as the functions have different number of parameters or different types for the parameters, and have the same return type. This way, compiler knows which version of the function is being called, based upon the arguments being passed.

```
int max (int a,int b); //return the maximum among two
int max (int a,int b, int c); //return the maximum among three
double average (int a, int b);
double average (double a, double b);
```

Variables with same name. You can declare multiple variables with same name, as long as they do not have the same scopes.

7 Function Design Exercises

Review the principles of function design (black box analogy, input-output, information hiding, self-contained and well-defined functionalities, minimum coupling). A function should do one and only one thing well, so that it can be reused in as many as situations as possible. Now critique the following function design, and discuss how to improve their design.

1. Input-Output of Function

```
bool leap;

/* This function sets global variable leap to true if year is leap year,
    and false otherwise
*/
void LeapYear (int year)
{
   if (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0))
   {
    cout <<"It's a leap year\n";
    leap = true;
  }
  else
  {
    cout <<"it's not a leap year\n";
```

```
    leap = false;
  }
}
```

2. Parameter v.s. Global Variables v.s. Local Variables

```
bool leap;

// Return the number of days in a month. If month is 2,
// check global variable leap to see if  it's leap year or not .
int DaysInMonth (int month, int MaxDays)
{
   if (month==1 || month == 3 || month == 5 || month == 7 || month == 8
      || month == 10 || month == 12)
    MaxDays = 31;
  else if (month == 4 || month == 6 || month == 9 || month == 11)
    MaxDays = 30;
  else if (leap && month == 2)
    MaxDays = 29;
  else if (!leap && month == 2)
    MaxDays = 28;
  return MaxDays;

}
```

3. Pre-conditions Checking

```
#include <assert.h>

/*
 * Calculates number of days in the year for a given date
 * Precondition: month, day, and year specifies a valid date
 * Postcondition: returns the number of days passed since the
 * beginning of the year to the given date
 */
int DaysInYear (int month, int day, int year)
{
   assert (ValidDate (month, day, year);

   ...
```

```
}
```

4. Post-conditions verification: Does the following function fufill its contract?

```
/*
 * Return true of the given date is valid, otherwise return false
 * Precondition: none
 * Postcondition: return true if month/day/year is a valid date,
 *                return false otherwise.
 */
bool IsValidDate (int month, int day, int year)
{
   if (year<0 || month>12 || month<0)
  return false;
   if ( (month==4||month==6||month==9||month==10) && (day<=30 && day>=1) )
  return true;
   if (month==2 && IsLeapYear(year)==true)
  return true;
   else
  return false;
}
```